(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 959 409 A1

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
24.11.1999 Bulletin 1999/47

(51) Int. Cl.[6]: G06F 12/02

(21) Application number: 99302913.1

(22) Date of filing: 15.04.1999

(84) Designated Contracting States:
AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU MC NL PT SE
Designated Extension States:
AL LT LV MK RO SI

(30) Priority: 19.05.1998 US 81359

(71) Applicant:
Northrop Grumman Corporation
Los Angeles, California 90067-2199 (US)

(72) Inventors:
- Willard, Brian Alan
Palm Bay, Florida 32907 (US)
- Frieder, Ophir
Melbourne, Florida 32901 (US)

(74) Representative:
Maury, Richard Philip et al
Sommerville & Rushton,
Business Link Building,
45 Grosvenor Road
St. Albans, Herts AL1 3AW (GB)

(54) Dynamic memory reclamation without compiler or linker assistance

(57) A method of dynamic memory reclamation for reclaiming memory leaks is disclosed, in which inaccessible heap segments are reclaimed without recompilation or relinkage of an application. The method comprises intercepting a call from an application to allocate or deallocate a heap segment, forwarding the call to the memory manager, passing the memory allocation and deallocation information from the memory manager to a file, performing a conservative garbage collection process using the memory allocation and deallocation information in the file to detect inaccessible heap segments, and calling to the memory manager from the file to deallocate the inaccessible heap segments. These steps are repeated periodically.

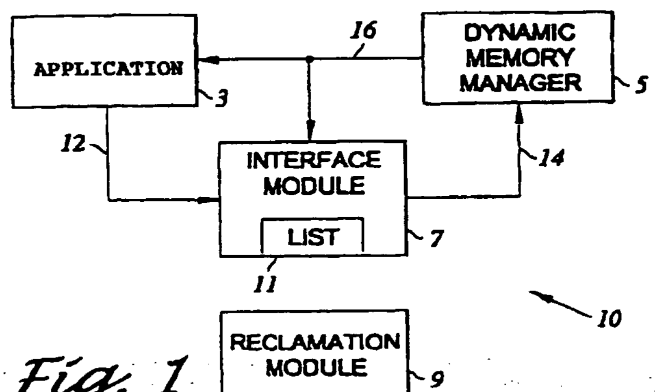


Fig. 1

EP 0 959 409 A1

Description

Field of the Invention

[0001] This invention relates generally to a method for dynamic memory management in a computer, and more particularly to a method of dynamic memory reclamation that detects and reclaims memory leaks without recompiling or relinking an application.

Background of the Invention

[0002] Computer systems may manage computer memory dynamically. Dynamic memory management refers to the process by which blocks of memory are allocated temporarily for a specific purpose and then deallocated when no longer needed for that purpose. Deallocated blocks are available for reallocation for another purpose. The process that dynamically manages the memory is referred to as the memory manager. The memory that the memory manager manages is referred to as a "heap." When an application program needs a block of memory to store data, the program sends a request to the memory manager. The memory manager allocates a block of memory in the heap to satisfy the request and sends a pointer to the block of memory to the program. The program can then access the block of memory through the pointer.

[0003] In the case of programs written in certain languages, such as C++, blocks of memory can be allocated automatically or dynamically. Automatic memory blocks are automatically allocated when a procedure is entered and automatically deallocated when the procedure is exited. Conversely, dynamic memory blocks are allocated by an explicit call to the memory manager and deallocated by either an explicit call to the memory manager or automatically through a memory reclamation technique known as garbage collection. Typically, automatically allocated memory blocks are stored in a stack and dynamically allocated memory blocks are stored in a heap.

[0004] A program can only access a dynamic memory block through a pointer. A pointer is a memory location that contains the address of an allocated (or used) heap segment. If the program overwrites a pointer, then the corresponding heap segment becomes "inaccessible" to the program. An allocated heap segment may be pointed to by several pointers, located on the stack or in another allocated heap segment. Only when all the pointers are overwritten, or are part of another inaccessible heap segment, does the heap segment become inaccessible. A program cannot retrieve data from or write data to an inaccessible heap segment. These inaccessible allocated heap segments are known as memory leaks.

[0005] For efficient memory usage, inaccessible heap segments must be "reclaimed" so that they can be reallocated. The identification and reclaiming of inaccessible heap segments is known as garbage collection. A system or method for performing garbage collection is referred to as a garbage collector. The first garbage collector developed was the reference counting algorithm. It is based on counting the number of active references to dynamically allocated objects to determine which objects are inaccessible. The mark/sweep algorithm was developed at about the same time. Mark/sweep is a tracing algorithm and relies on global traversal of all allocated objects to determine the inaccessible objects.

[0006] Another garbage collector is the copying algorithm. This collector divides the heap equally into two parts. One part contains current data and the other old data. The algorithm traverses all active objects in the current data part, copies them to the other part, leaving the inactive objects uncopied. After all active objects have been traced, the roles of the two parts are swapped. Generational and incremental garbage collectors are designed to improve the performance of garbage collection, making it feasible in real-time applications. The idea behind generational garbage collection is that most objects die young. The algorithm segregates objects by age in two or more generations of heap, and concentrates effort on reclaiming those objects most likely to be garbage, i.e., young objects. Incremental algorithms decrease the length of garbage collection pauses by interleaving small amounts of collection with the real-time application program's execution. Many different embodiments of these collectors can be found in the prior art. However, implementing these collectors requires recompiling or relinking the application. Thus there is a need for a system and method for dynamic memory reclamation without recompiling or relinking the application program.

[0007] Servers in large scale information retrieval systems cannot long endure even small memory leaks because of their massive input/output demands. Memory leaks are significantly enhanced because information retrieval systems demand massive caching of massive data. A small memory leak per transaction could exhaust the available memory in a matter of days. Many existing information retrieval systems are implemented in C or C++. These languages do not automatically reclaim dynamically allocated memory, and some level of memory leaks inevitably go undetected despite high quality, meticulous programming. Thus, there is a need for a system and method for reclaiming memory leaks without programmer assistance.

[0008] There are many available tools to perform memory reclamation in the prior art. These tools require at least one of the following: compiler assistance, source code instrumentation, add-on utilities that must be linked with the application, and replacing elements in the language memory manager. Compiler assistance and source code instrumentation can be very effective in garbage collectors, however, source code and/or a build environment is often not available for a particular application. Relinking applications to include add-on

utilities may change the execution of an application, masking previously manifest defects or introducing new ones. Thus, there is a need for a system and method for reclaiming memory leaks without recompiling or relinking an application program.

Summary of the Invention

[0009] A method of dynamic memory reclamation for reclaiming memory leaks is disclosed, in which inaccessible heap segments are reclaimed without recompilation or relinkage of an application. The method comprises intercepting a call from an application to allocate or deallocate a heap segment, forwarding the call to the memory manager, passing the memory allocation and deallocation information from the memory manager to a file, performing a conservative garbage collection process using the memory allocation and deallocation information in the file to detect inaccessible heap segments, and calling to the memory manager from the file to deallocate the inaccessible heap segments. These steps are repeated periodically.

[0010] In the preferred embodiment, the conservative garbage collection process comprises a conservative mark/sweep garbage collector. A list of allocated heap segments is maintained in the file. The application stops when garbage collection tarts. Heap segments are conservatively scanned to locate pointers, values corresponding to addresses of allocated heap segments. The heap segments pointed to by pointers are marked on the list as accessible. The list is scanned and all unmarked heap segments on the list are considered inaccessible. The memory manager is instructed to deallocate the unmarked, inaccessible heap segments. The marked heap segments on the list are cleared, the deallocated heap segments are removed from the list, and the application resumes.

[0011] It is an object of the present invention to provide a method for dynamic memory management without recompiling or relinking to the application.

[0012] It is a further object of the present invention to provide a method for dynamic memory management in large scale information retrieval systems.

Brief Description of the Drawings

[0013]

Figure 1 represents an embodiment of the memory reclamation system of the present invention while an application program is running.

Figure 2 is a further representation of an embodiment of the system of the present invention while a conservative garbage collection process is running.

Figure 3 is a flow diagram that represents the preferred embodiment of the memory reclamation method of the present invention.

Detailed Description of the Preferred Embodiment

[0014] In a computer system, an application which dynamically allocates memory will send a request to the memory manager when it needs to allocate something to a heap segment, or to deallocate a heap segment. Referring to Figure 1, in an embodiment of the system 10 of the present invention, this request 12 is intercepted by an interface module 7 that emulates the interfaces of the memory manager 5. The application performs just as it would if the request was not intercepted. The interface module 7 forwards the request 14 to the memory manager 5, which performs the requested allocation or deallocation. The memory manager 5 performs just as it would if the request came directly from the application.

[0015] When the allocation or deallocation is completed, the memory manager 5 sends allocation/deallocation information 16 to the interface module 7 and to the application 3. The allocation/deallocation information 16 comprises the address and length of the heap segment. The interface module 7 uses the allocation/deallocation information to update a list of allocated heap segments 11. Allocated heap segments are added to the list and deallocated heap segments are removed from the list. The list is continuously updated while the application is running.

[0016] In a computer running an OpenVMS operating system, the memory manager would normally send the allocation/deallocation information to a module called the Heap Analyzer. The Heap Analyzer is a tool that presents application memory allocations to a programmer to identify areas where memory usage and performance can be improved. When the system of the present invention is implemented in OpenVMS, the interface module 7 replaces the Heap Analyzer and receives the allocation/deallocation information 16 from the memory manager 5. The system of the present invention may be implemented in a computer running a different operating system, e.g., Unix or Windows NT, by creating an interface module containing procedures that intercept all the memory manager routines.

[0017] A signal from the operating system scheduler tells the garbage collection algorithm to begin and the application program halts while the garbage collector runs. Referring now to Figure 2, the garbage collection process is performed by the reclamation module 9. The process detects inaccessible heap segments using the list of allocated heap segmenst 11 maintained by the interface module 7. The reclamation module 9 sends a request 20 to the memory manager 5 to deallocate the inaccessible heap segments. The memory manager 5 performs this function and sends the deallocation information 22 to the interface module 7. The interface module 7 then removes the deallocated heap segments from the list 11. Once this process is complete, the application 3 resumes and the system operates as described in reference to Figure 1. Thus inaccessible

heap segments are reclaimed without recompiling or relinking the application, since the application is blind to the garbage collection process and that process is conservative.

[0018] Halting an application for periods of garbage collection works well for applications with soft deadlines, e.g. information retrieval systems. However, real-time applications with hard deadlines cannot permit garbage collection in one action preempting application execution. To be suitable for real-time applications, the system may interleave small units of garbage collection with small units of application execution. Such an design choice is within the scope of the present invention.

[0019] Referring now to Figure 3, the preferred embodiment of the method of the present invention is described. While an application program 3 is running, it sends out allocation and deallocation requests 2. In the present method, these requests 2 are intercepted by an interface module 7. This module forwards the request 2a to the dynamic memory manager 5. The memory manager 5 performs the requested allocation or deallocation, and sends the resulting allocation/deallocation information 4 to the application 3 and back to the interface module 7. The interface module 7 maintains this information in a list of allocated heap segments 11. Allocated heap segments are added to the list and deallocated heap segments are removed from the list. These steps continue until the operating system scheduler signals the start of the garbage collection process and the halt of the application.

[0020] In the preferred embodiment of the present method, the garbage collection process comprises a conservative mark/sweep process. The mark/sweep process of the present invention differs from the traditional mark/sweep method by not delaying reclamation until all available heap storage is exhausted, nor does it reclaim garbage immediately. Instead, the process operates on a periodic reclamation cycle. The mark/sweep process has two main steps. The mark step partitions all allocated heap segments into two sets, accessible heap segments and inaccessible heap segments. The sweep step then frees all inaccessible heap segments for reuse by the application.

[0021] At a signal from the operating system scheduler 1, the process begins with the mark step, in which the reclamation module 9 scans the user stack, statically allocated memory, and accessible allocated heap segments for pointers. The reclamation module only scans these three areas of memory because it is assumed that pointers reside in memory that is accessible to the active application. Accessible allocated heap segments are scanned because they may contain pointers to other allocated heap segments. Each memory location in these areas is checked to determine if its value corresponds to an address within the bounds of an allocated heap segment, i.e. whether the value could be a pointer. The scan is conservative, in that the reclamation module receives no assistance from the compiler to determine whether a value is a pointer. Any word that could be a pointer (value equal to an address of an allocated heap segment) is treated like a pointer. An allocated heap segment that is pointed to by a pointer is considered accessible. Since the scan is conservative, it is possible that some inaccessible locations will be marked as accessible. Even so, the conservative scan works well since in practice the likelihood of properly aligned values having bit patterns of pointers is low.

[0022] Each accessible allocated heap segment is marked as such on the list of allocated heap segments 11. In the preferred embodiment, the list comprises a red-black binary tree structure that has a node for each allocated heap segment. Each node contains information about a heap segment including address, size, and a liveness mark if the heap segment is accessible.

[0023] The reclamation module then performs the sweep step, which frees the inaccessible heap segments. In the preferred embodiment the sweep step separates the marked and unmarked nodes in the binary tree, wherein the unmarked nodes correspond to inaccessible heap segments. Each node is visited in turn, and the addresses of all of the unmarked nodes are pushed onto the stack and the marked nodes are cleared. After all the nodes have been visited, the addresses of the unmarked heap segments are popped off the stack and the reclamation module sends a request 6 to the interface module to deallocate these inaccessible heap segments, thus reclaiming the memory leaks without rebuilding (recompiling or relinking) the application. The interface module forwards this request to the dynamic memory manager. It is unknown to the memory manager from where it is invoked, so it operates as if the application had called it to deallocate heap segments. The deallocation information generated by the memory manager is sent to the interface module. The nodes corresponding to the deallocated heap segments are removed from the list of allocated heap segments, the garbage collection process halts, and the application resumes.

[0024] During the garbage collection process, the reclamation module records accessible and inaccessible heap segments that are detected. At the start of the process, the reclamation module records a time stamp in a log file. During the mark step, as pointers are found, an entry is made in the log file which records the accessible heap segment address, size, and mark, the pointer value and the program counter of the instruction in the application that allocated the heap segment.

[0025] During the sweep step, as inaccessible heap segments are identified, an entry is made in the log file which records the inaccessible heap segment address, size, and mark, the start address of the heap segment and the program counter of the instruction in the application that allocated the heap segment. At the end of the garbage collection process, the redamation module records the number of inaccessible heap segments detected during the process.

[0026] The time cost of the mark step of the garbage collection process is proportional to the number of memory locations in the stack, statically allocated data, and accessible allocated heap segments, since each location must be scanned for pointers. The time cost of the sweep step is proportional to the total number of allocated heap segments, since each allocated heap segment has a node on the list that must be visited. The total time cost is the sum of the costs of the mark and the sweep steps, and depends on the processing speed of the computer.

[0027] In the preferred embodiment, the garbage collection process is performed periodically, for example every 10 seconds. During initialization the reclamation module requests the operating system to schedule the garbage collection process every 10 seconds. The periodicity of the garbage collection process may be adjusted without going beyond the scope of the present invention.

[0028] It is to be understood that the described embodiments are merely illustrative of the many specific embodiments which represent applications of the present invention. Other embodiments may be readily devised by those skilled in the art without departing from the spirit and scope of the present invention.

Claims

1. A method for dynamic memory reclamation in a computer having a memory containing a heap, and at least one of an operating system with a dynamic memory manager and a language runtime environment with a dynamic memory manager, wherein an application program sends requests for memory allocation and deallocation to the dynamic memory manager, said method comprising the steps of:

   a) intercepting a request for memory allocation or deallocation from the application to the dynamic memory manager, wherein the request is intercepted by a module which emulates interfaces of the dynamic memory manager;
   b) forwarding the request to the dynamic memory manager, which then performs the requested memory allocation or deallocation, wherein memory allocation/deallocation information is created;
   c) communicating the memory allocation/deallocation information to the module, wherein the module maintains a list of allocated heap memory segments;
   d) performing a conservative garbage collection process using the list of allocated heap memory segments in the module to detect inaccessible heap memory segments; and
   e) sending a reclamation request to the dynamic memory manager from the module to deallocate the inaccessible heap memory segments, whereby the inaccessible heap memory segments are detected and deallocated without rebuilding the application.

2. The method of claim 1 wherein the garbage collection process is performed periodically.

3. The method of claim 1 wherein the conservative garbage collection process comprises a conservative mark/sweep garbage collection process.

4. The method of claim 3 wherein the conservative mark/sweep garbage collection process comprises the steps of:

   a) locating memory locations that contain values that correspond to addresses of allocated heap memory segments, wherein each address conservatively identifies an accessible allocated heap memory segment;
   b) marking each accessible allocated heap memory segment on the list of allocated heap memory segments;
   c) scanning the list of allocated heap memory segments; and
   d) identifying unmarked heap memory segments as inaccessible heap memory segments.

5. A dynamic memory reclamation system in a computer having a memory, and at least one of an operating system with a dynamic memory manager and a language runtime environment with a dynamic memory manager, wherein an application program sends requests to the dynamic memory manager to allocate and deallocate heap segments, said reclamation system comprising:

   a) an interface module that intercepts the requests from an application and forwards the requests to the dynamic memory manager;
   b) memory allocation and deallocation information generated by the dynamic memory manager in response to the requests, the memory allocation and deallocation information being sent to the interface module and to the application;
   c) wherein the interface module maintains a list of allocated heap segments;
   d) a relcamation module, which performs a conservative garbage collection process using the list of allocated heap segments to detect inaccessible heap segments, and sends a deallocation request to the interface module to deallocate the inaccessible heap segments, wherein the interface module forwards the deallocation request to the dynamic memory

manager whereby the inaccessible heap segments are reclaimed without rebuilding the application.

6. The dynamic memory management system of claim 5 wherein the conservative garbage collection process comprises a conservative mark/sweep garbage collector.

7. The dynamic memory management system of claim 5 wherein the reclamation module performs the conservative garbage collection process periodically.

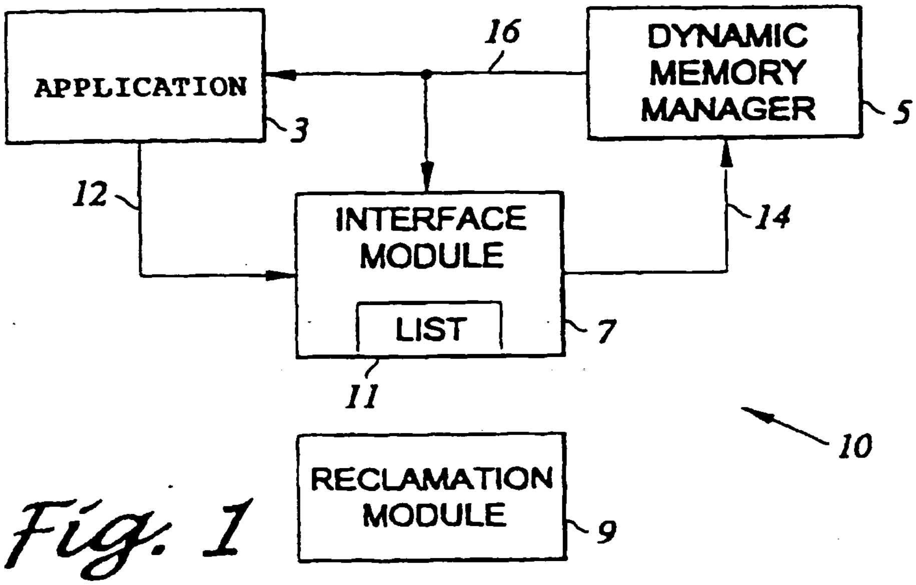

APPLICATION
3
16
DYNAMIC MEMORY MANAGER
5
12
INTERFACE MODULE
LIST
7
11
14
RECLAMATION MODULE
9
10

Fig. 1

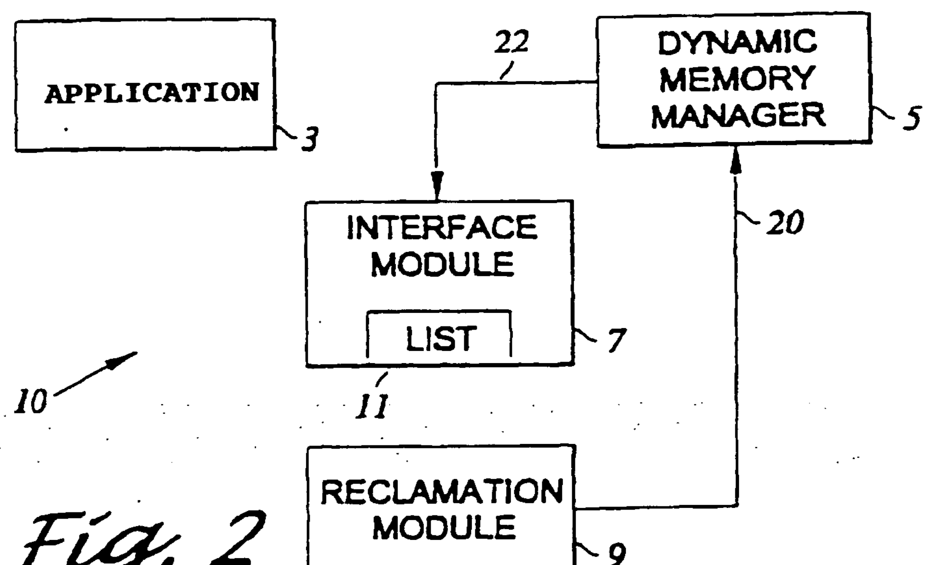
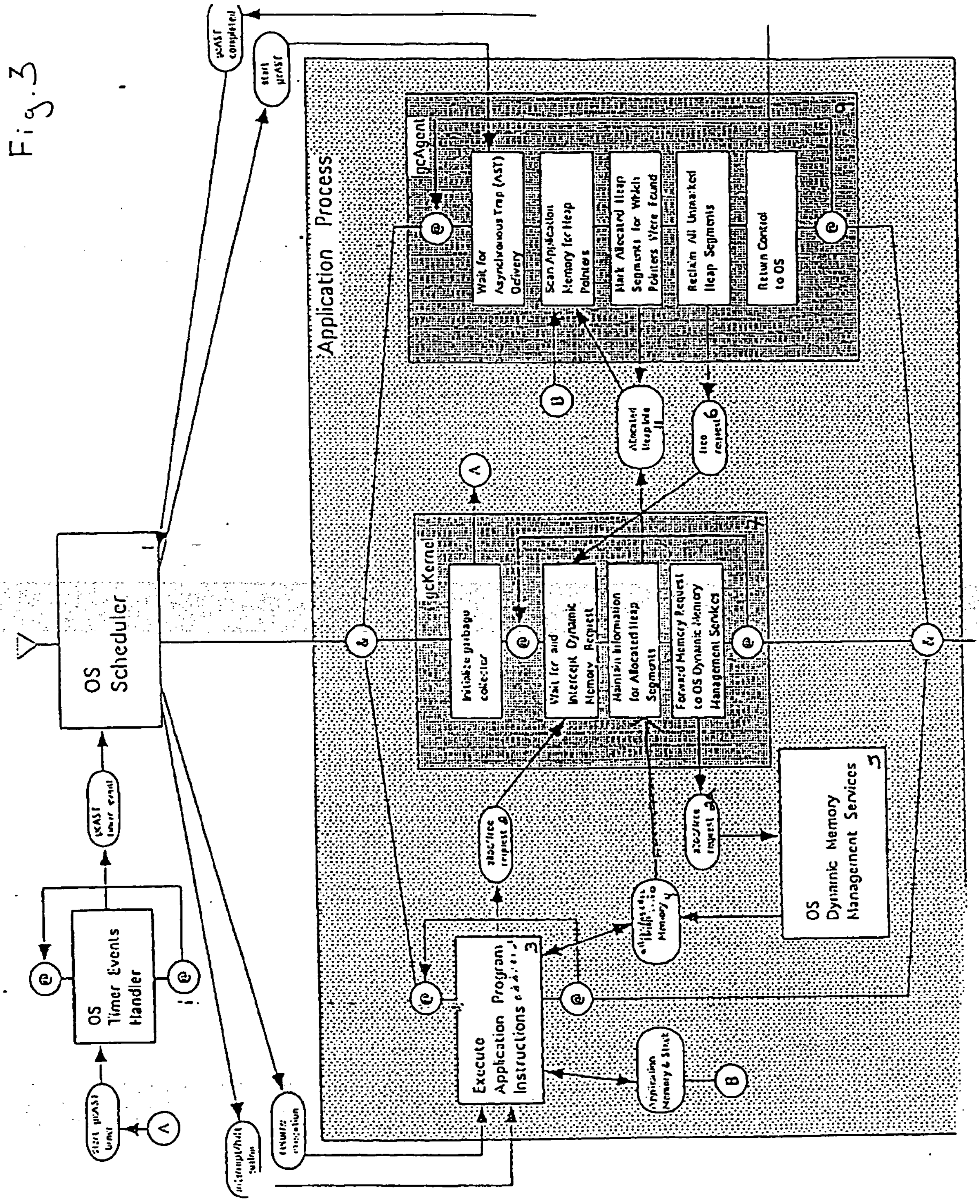

APPLICATION
3
22
DYNAMIC MEMORY MANAGER
5
INTERFACE MODULE
LIST
7
11
20
10
RECLAMATION MODULE
9

Fig. 2

Fig. 3


OS Scheduler
OS Timer Events Handler
Application Process
gcKernel
Initialize garbage collector
Wait for and Intercept Dynamic Memory Request
Maintain Information for Allocated Heap Segments
Forward Memory Request to OS Dynamic Memory Management Services
OS Dynamic Memory Management Services
Execute Application Program Instructions
gcAgent
Wait for Asynchronous Trap (AST) Delivery
Scan Application Memory for Heap Pointers
Mark Allocated Heap Segments for Which Pointers Were Found
Reclaim All Unmarked Heap Segments
Return Control to OS

European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 99 30 2913

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.6)
X	RODRIGUEZ-RIVERA G ET AL: "NONINTRUSIVE CLONING GARBAGE COLLECTION WITH STOCK OPERATING SYSTEMSUPPORT" SOFTWARE PRACTICE & EXPERIENCE, vol. 27, no. 8, 1 August 1997 (1997-08-01), pages 885-904, XP000693733 ISSN: 0038-0644 * page 892, line 13 - page 894, line 33 * ---	1-7	G06F12/02
A	BOEHM H -J ET AL: "GARBAGE COLLECTION IN AN UNCOOPERATIVE ENVIRONMENT" SOFTWARE PRACTICE & EXPERIENCE, vol. 18, no. 9, 1 September 1988 (1988-09-01), pages 807-820, XP000001796 ISSN: 0038-0644 * page 816, line 4 - page 818, line 6 * -----	1-7	
			TECHNICAL FIELDS SEARCHED (Int.Cl.6)
			G06F

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	31 August 1999	Nielsen, O

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)